

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21

Sesión S02: Drivers

Sargent, es horrible, Sargent...

Al final empezaron a encontrarnos a todos... Nos están arreglando en todas partes... Ellos... incluso están añadiendo... una nueva característica



Les mostraré... Quiero que dos "bugs" valientes ataquen a la nueva característica



Automatización de las pruebas

- Objetivo: ejecutar de forma automática todos los casos de prueba previamente diseñados
- Como resultado obtendremos un informe (que revelará la presencia de defectos en el código)

Pruebas unitarias

- El código a probar se denomina **SUT**.
- En nuestro caso **SUT** representa **una unidad**.
- Hemos definido **una unidad** como un **método java**

Pruebas de unidad dinámicas: drivers

- Un **driver** es un **código que permite ejecutar un caso de prueba de forma automática**.
- Necesitamos ejecutar código para **detectar defectos (pruebas dinámicas)**

Implementación de drivers: JUnit 5

- **Librería para implementar y ejecutar las pruebas API**

Ejecución de los tests unitarios (drivers) con **Maven**

Vamos al laboratorio...

P

AUTOMATIZACIÓN DE LAS PRUEBAS

(se trata de implementar código (drivers) para ejecutar los tests de forma automática)

P

ACTIVIDADES DEL PROCESO DE PRUEBAS

PLANIFICACIÓN Y CONTROL

DISEÑO

AUTOMATIZACIÓN

EVALUACIÓN

Pruebas dinámicas: pruebas en las que para contestar a esa pregunta necesariamente necesitamos ejecutar el código

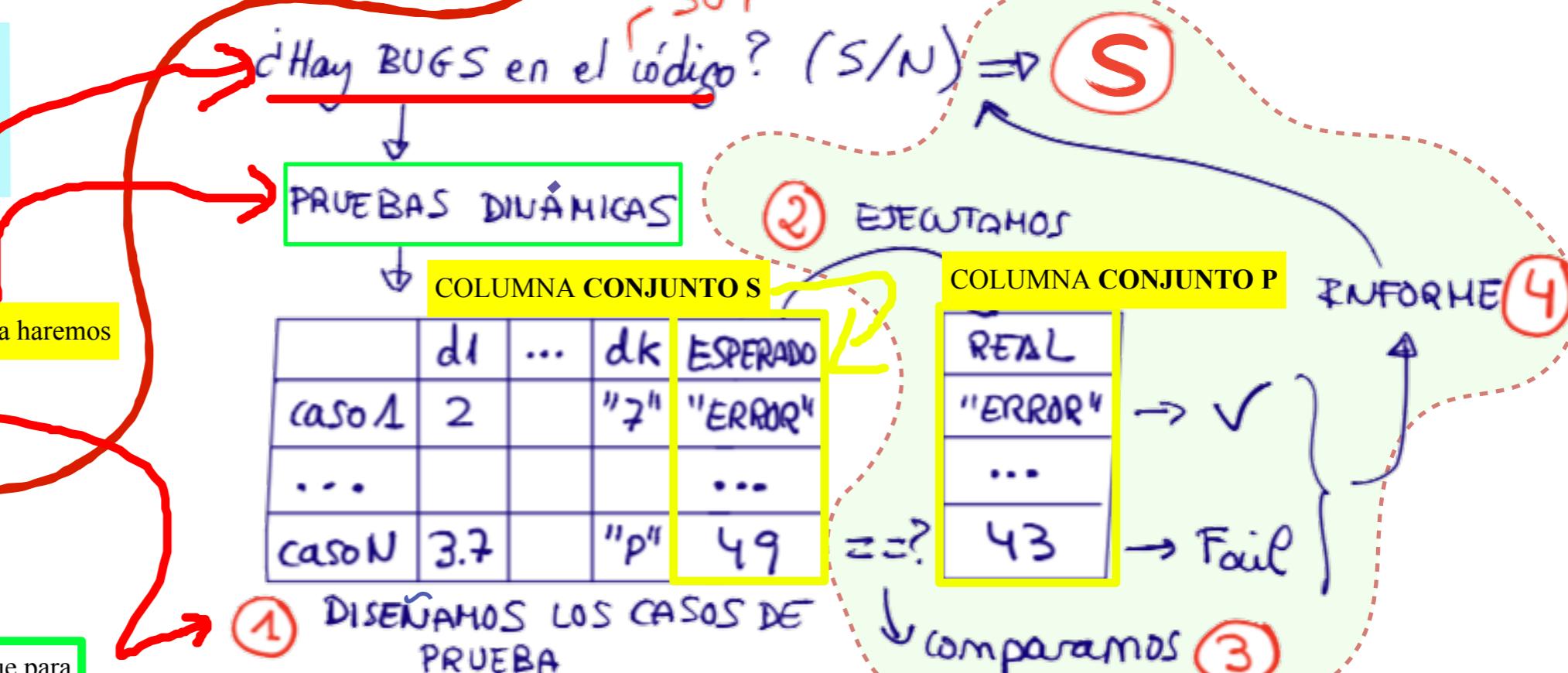
Para eso nuestro código de pruebas tendrá que:

- Una vez establecidas las precondiciones sobre los valores de entrada:
- Proporcionar los datos de entrada + resultado esperado (1) al código a probar (SUT)
- Obtener el resultado real (2)
- Comparar el resultado esperado con el real (3)
- Emitir un informe que contestará a nuestra pregunta inicial (4)

ALGORITMO DRIVER → Diapositiva 4

CÓDIGO QUE VAYAMOS A PROBAR

→ **SUT** = System Under Test



El **OBJETIVO** es poder realizar los procesos 2, 3 y 4 "pulsando un botón"

A dicho código de pruebas lo llamaremos **DRIVER**.

Implementaremos tantos drivers como casos de prueba.

Cada driver invocará a nuestra **SUT** y proporcionará un informe

PRUEBAS UNITARIAS

P EL CODIGO MÁS PEQUEÑO SOBRE EL QUE VOY A HACER PRUEBAS S

Sintácticamente, una **unidad de programa** es una "pieza" de código, que **puede ser invocada desde fuera** de la unidad **y puede invocar a otras** unidades de programa

Una unidad de programa **implementa una función** bien definida, y proporciona un nivel de abstracción para la implementación de funcionalidades de mayor nivel

Las pruebas unitarias son realizadas por los propios **programadores**. Éstos necesitan

VERIFICAR que el código funciona **correctamente** (tal y como se esperaba)

Hasta que el programador **no implemente** la **unidad** y esté **completamente probada**, el código fuente de una unidad no se pone a disposición del resto de **miembros** del grupo (normalmente a través de un sistema de control de versiones)

Pueden realizarse pruebas unitarias de forma **estática** y/o **dinámica**

→ para detectar errores necesitamos ejecutar código!!!

Objetivo: encontrar **DEFECTOS** en el código de las **UNIDADES** probadas

Nuestra **SUT** será una **Unidad** **unidad 1**

unidad 2

unidad 3

unidad 4

unidad 5

unidad 6

unidad 7

CADA PIEZA ES UNA UNIDAD

PROBLEMA: ¿COMO PROBAR UNA UNIDAD SIN ESTAR "ENGANCHADO" AL APARATO FINAL?

Queremos probar cada UNIDAD por SEPARADO!



La **CUESTIÓN** fundamental será cómo AISLAR el código de cada unidad a probar

PRUEBAS DE UNIDAD DINÁMICAS: DRIVERS

Requieren ejecutar una unidad de forma AISLADA para poder detectar defectos en el código de dicha unidad

SI NOS EQUIVOCAMOS A LA HORA DE IMPLEMENTAR UN DRIVER NO NOS SERVIRÁ DE NADA Y EL SUT NO SE PODRÁ PROBAR CORRECTAMENTE

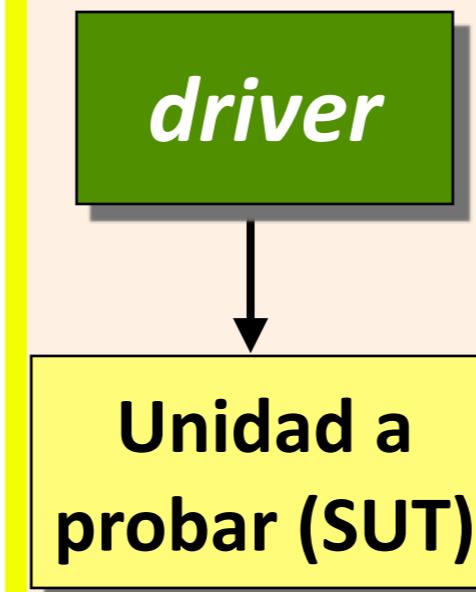
El "tamaño" de las unidades dependerá de lo que consideremos como unidad.

En nuestro caso concreto, vamos a definir una UNIDAD como un MÉTODO JAVA

Cada unidad será invocada desde un driver durante las pruebas, con los datos de entrada diseñados previamente

HACERLO DE LA FORMA MÁS SENCILLA

```
1. //algoritmo de un driver
2. informe driver() {
3.   d= prepara_datos_entrada();
4.   esperado= resultado Esperado;
5.   //invocamos al SUT
6.   real= SUT(d);
7.   //comparamos el resultado
8.   //real con el esperado
9.   c= (esperado == real);
10.  informe= prepara_informe(c);
11.  return informe;
12. }
```



driver : conductor de la prueba.
Contiene el código necesario para EJECUTAR el caso de prueba sobre SUT

Unidad a probar (SUT)

CÓDIGO DE UN MÉTODO JAVA, COMPROBAR SI ALGUNA DE LAS LINEAS ESTÁ MAL ESCRITA

SUT : es el código que queremos probar. En este caso, representa a una unidad

En esta sesión vamos a ver cómo implementar drivers con JUnit para ejecutar pruebas unitarias dinámicas !!!!

JUNIT 5

<https://junit.org/junit5/docs/current/user-guide/>

JUnit es un API java que permite implementar los drivers y ejecutar los casos de prueba sobre componentes (SUT) de forma automática

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Módulo que proporciona APIs relacionados con la "detección" y ejecución de los tests desde OTRAS APLICACIONES EXTERNAS (p.e. maven o IntelliJ)

Y COMPILACIÓN
Módulo que proporciona APIs para implementar y ejecutar los tests JUnit5

SOLO IMPLEMENTAR, NO EJECUTAR DESDE CÓDIGO, ESO LO HARÁ MAVEN E INTELLIJ

Módulo que proporciona APIs que para ejecutar tests VERSIONES PREVIAS

- JUnit se puede utilizar para implementar drivers de pruebas unitarias y también de integración Y OTROS TIPOS DE PRUEBAS
 - En una prueba unitaria estamos interesados en probar una única unidad (nuestra SUT será un método Java)
 - En una prueba de integración estamos probando varias unidades (la SUT en este caso representará a un subconjunto de unidades)



No es suficiente con conocer el API JUnit, hay que usarlo correctamente, siguiendo unas normas !!!

¿CÓMO IDENTIFICAMOS UN DRIVER CON JUNIT 5?

driver = test JUnit = método anotado con @Test

- Un **driver** automatiza la ejecución de un **caso de prueba**.

JUnit denomina test (uno por cada caso de prueba) a un **MÉTODO** sin parámetros, que devuelve **void**, y está anotado con **@Test** (@org.junit.jupiter.api.Test)

```
package mismo.paquete.claseAprobar;  
  
import org.junit.jupiter.api.Test;  
  
class TrianguloTest {  
  
    @Test  
    void C01testQueNoHaceNada() {  
    }  
}
```

Los tests deben agruparse lógicamente con el **SUT correspondiente** (deben pertenecer al **mismo PAQUETE**)

La **CLASE** de pruebas tendrá el **mismo nombre que la clase que contiene el SUT precedida (o seguida) por "Test"**

El **DRIVER** será un **método sin parámetros que devuelve "void"** y está **anotado con @Test**

Cada método anotado con **@Test** implementará un driver para **un ÚNICO caso de prueba**!!!!

No es necesario que la clase de pruebas ni los tests sean **"public"** !!

IMPLEMENTACIÓN DE UN DRIVER (I)

Cualquier driver sigue el mismo algoritmo:

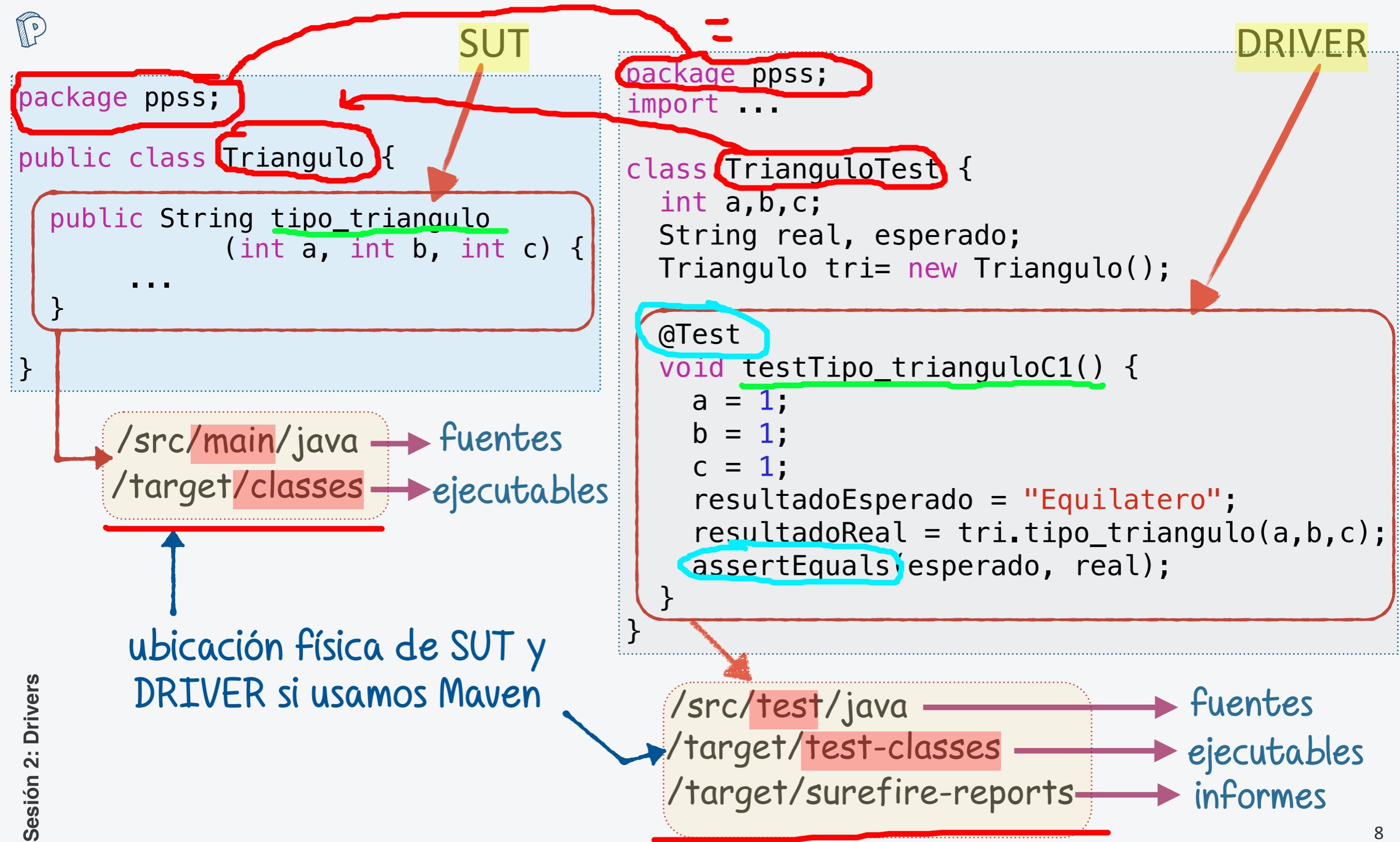
```
//algoritmo de un driver
informe driver() {
    d= prepara_datos_entrada();
    esperado= resultado Esperado;
    real= SUT(d); //llamamos a SUT
    c= (esperado == real); //comparamos el resultado real con el esperado
    informe= prepara_informe(c);
    return informe;
}
```

OJO.- CÓDIGO ADICIONAL, SI LO HACEMOS MAL, APAGA Y VÁMONOS

para implementar un driver necesitamos
diseñar los casos de prueba previamente

IMPLEMENTACIÓN DE UN DRIVER(II)

P El código de pruebas está físicamente separado del código fuente del SUT



SENTENCIAS ASSERT (org.junit.jupiter.api.Assertions)

- Junit proporciona sentencias (aserciones) para determinar el resultado de las pruebas y poder emitir el informe correspondiente
- Son métodos estáticos, cuyas principales características son:
 - Se utilizan para comparar el resultado esperado con el resultado real
 - El orden de los parámetros para los métodos assert... es:
 - resultado ESPERADO, resultado REAL [, mensaje opcional]

APARECE SI EL ESPERADO != REAL

```
@Test
void standardAssertions() {
    /*todas las aserciones presentan
     estas tres variantes: */
    assertEquals(2, 2);
    assertEquals(4, 4, "Mensaje opcional");
    assertEquals(8, 8, () -> "Mensaje creado"
        RE, RR, F(X) LAMBDA + "en tiempo de ejecución");
}
```

ELI

Podemos usar los métodos directamente:

```
import org.junit.jupiter.api.Assertions;
...
Assertions.assertEquals(...);
```

O referenciarlos mediante un import estático:

```
import static
org.junit.jupiter.api.Assertions.*;
...
assertEquals(...);
```

Todos los métodos "assert" generan una excepción de tipo AssertionFailedError si la aserción no se cumple!!!

Un test puede requerir varias aserciones

P

AGRUPACIÓN DE ASERCIÓNES

S

Qué ocurre cuando un test contiene varias aserciones??

Dado que un test termina en cuanto se lanza la primera excepción (no capturada), en el caso de que nuestro test contenga varias aserciones, usaremos el método **assertAll**, para agruparlas. Este caso, se ejecutan todas, y si alguna falla se lanza la excepción **MultipleFailuresError**

assertAll

```
public static void assertAll(String heading,
                             Executable... executables)
throws MultipleFailuresError
```



Asserts that *all* supplied executables do not throw exceptions.

```
// Añadimos un elemento a una colección llena
@Test
public void testC3Add() {
    int[] arrayEsperado = {1,2,3,4,5,6,7,8,9,10};
    int numElemEsperado = 10;

    int[] arrayEstadoInicial = Arrays.copyOf(arrayEsperado, arrayEsperado.length);
    DataArray colección = new DataArray(arrayEstadoInicial, 10);
    colección.add(11);
    //Agrupamos las aserciones. SE EJECUTAN TODAS siempre
    assertAll("GrupoTestC3",
              ()-> assertArrayEquals(arrayEsperado, colección.getColección()),
              ()-> assertEquals(numElemEsperado, colección.size())
    ); //Se muestran todos los "fallos" producidos
}
```



Esta opción nos proporciona más información!!

```
// Añadimos un elemento a una colección llena
@Test
public void testC3Add() {
    ...
    //No agrupamos las excepciones. Si falla la primera, la segunda NO se ejecuta
    assertArrayEquals(arrayEsperado, instance.getColección());
    assertEquals(numElemEsperado, instance.size()); ←
    SI FALLA EL PRIMER ASSERT, NO PODEMOS VER SI FUNCIONA EL SIGUIENTE
}
```



Sólo se ejecuta si el assert anterior no falla

PRUEBAS DE EXCEPCIONES

assertThrows()

assertThrows

```
public static <T extends Throwable> T assertThrows(Class<T> expectedType,  
Executable executable, String message)
```

Assert that execution of the supplied executable throws an exception of the expectedType and return the exception.

If no exception is thrown, or if an exception of a different type is thrown, this method will fail.

If you do not want to perform additional checks on the exception instance, simply ignore the return value.

Fails with the supplied failure message.

- Si el resultado esperado es que nuestro SUT lance una excepción, usaremos la aserción assertThrows()

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertThrows;  
  
@Test  
void exceptionTesting() {  
    //si sut() lanza la excepción de tipo ExpectedException  
    //asignamos la excepción a la variable "exception"  
    ExpectedException exception = assertThrows(ExpectedException.class,  
        () -> sut(e1,e2));  
    DEVUELVE UNA EXCEPCIÓN  
    //mostramos el mensaje asociado a la excepción  
    assertEquals("a message", exception.getMessage());  
}
```

Si al ejecutar sut()
NO se lanza la
excepción de tipo
ExpectedException,
la ejecución del
test fallará.

Si solamente queremos comprobar que se lanza la
excepción, esta sentencia NO es necesaria

PRUEBAS DE EXCEPCIONES

assertDoesNotThrow()

assertDoesNotThrow

```
@API(status=STABLE, since="5.2") public static void assertDoesNotThrow(Executable executable,  
String message)
```

Assert that execution of the supplied executable does *not* throw any kind of exception.

Usage Note

Although any exception thrown from a test method will cause the test to *fail*, there are certain use cases where it can be beneficial to explicitly assert that an exception is not thrown for a given code block within a test method.

Fails with the supplied failure message.

- Si el resultado esperado es que nuestro SUT no lance ninguna excepción, usaremos la aserción `assertDoesNotThrow()`

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;  
  
@Test  
void NotExceptionTestingV1() {  
    //si sut() lanza una excepción el test fallará  
    assertDoesNotThrow(() -> sut(e1,e2), "Excepción lanzada"));  
}
```

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;  
  
@Test  
void NotExceptionTestingV2() throws ExpectedException{  
    sut(e1,e2);  
}
```

CHECKED HEREDA DE ASSERTION

```
void sut(Type1 param1, Type2 param2) throws ExpectedException
```

No necesitamos capturar la excepción. El código queda más "limpio!"

Debemos evitar en lo posible esta alternativa. Ya que si se lanza la excepción el resultado del test será ERROR, en lugar de FAILURE

P ANOTACIONES



@BeforeEach, @AfterEach, @BeforeAll, @AfterAll

- En el caso de que **TODOS** los tests requieran las **MISMAS** acciones para preparar los datos de entrada (antes de ejecutar el elemento a probar), implementaremos dichas acciones comunes en un método anotado con **@BeforeEach**.
 - De esta forma **reduciremos la duplicación de código** y nos aseguraremos de que todos los tests **parten del mismo estado inicial**
- De igual forma, usaremos la anotación **@AfterEach** en un método que contenga todas las acciones comunes a realizar **después de la ejecución de CADA test** (por ejemplo, podríamos necesitar asegurarnos de que una conexión por socket esté cerrada después de ejecutar cada test)

@BeforeEach y **@AfterEach** se usan con métodos void SIN parámetros!!!

- Si es necesario realizar acciones previas a la ejecución de **TODOS** los tests una **ÚNICA VEZ**, (o después de ejecutar todos los tests), implementaremos dichas acciones en un método anotado con **@BeforeAll** o **@AfterAll**, respectivamente.

@BeforeAll y **@AfterAll** se usan con métodos estáticos void SIN parámetros!!!

- Estas anotaciones permiten **initializar** y **restaurar** el entorno en el que se ejecuta cada test o cada conjunto de tests, de forma que si algún test (o conjunto de tests) "alteran" dicho estado, el siguiente test/conjunto de tests pueda ejecutarse normalmente con independencia del resultado de la ejecución de tests anteriores.

NO debemos implementar tests cuya ejecución dependa del resultado de ejecutar ningún otro test!!!



EJEMPLO @BeforeEach, @AfterEach, @BeforeClass, @AfterClass

```
P S  
P import org.junit.jupiter.api.AfterAll;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
class OutputTest {  
    static File output;  
  
    @BeforeEach void createOutputFile() {  
        output = new File(...);  
    }  
    @AfterEach void deleteOutputFile() {  
        output.delete();  
    }  
    @BeforeAll static void initialState() {  
        //initial code  
    }  
    @AfterAll static void finalState() {  
        //final code  
    }  
    @Test void test1WithFile() {  
        // code for test case objective  
    }  
    @Test void test2WithFile() {  
        // code for test case objective  
    }  
}
```

ORDEN de ejecución asumiendo que
test1Withfile() se ejecuta ANTES de
test2WithFile()

1. initialState()
2. createOutputFile()
3. test1WithFile()
4. deleteOutputFile()
5. createOutputFile()
6. test2WithFile()
7. deleteOutputFile()
8. finalState()

No debemos asumir
ningún orden de
ejecución de nuestros
tests!!!



P

ETIQUETADO DE LOS TESTS: @Tag

S Permiten SELECCIONAR la ejecución de un subconjunto de tests

- Tanto las clases como los tests pueden anotarse con @Tag. Esta anotación permite "etiquetar" nuestros tests para FILTRAR su "descubrimiento" y "ejecución"
 - Si anotamos la clase, automáticamente estaremos etiquetando todos sus tests
 - Podemos usar varias "etiquetas" para la clase y/o los tests

```
import org.junit.jupiter.api.Tag;  
import org.junit.jupiter.api.Test;  
  
{@Tag("fast")  
{@Tag("model")  
class TaggingDemo {  
  
    @Test  
    @Tag("taxes")  
    void testingTaxCalculation() {  
        ...  
    }  
}}
```

Este test está etiquetado como "fast", "model" y "taxes"

Las anotaciones @Tag nos permitirán DISCRIMINAR la ejecución de los tests según sus etiquetas.

Ejemplos de etiquetas:

- "Firefox", "Explorer", "Safari", ...
- "Windows", "OSX", "Ubuntu", ...
- "Unitarios", "Integracion", "Sistema", ...

TESTS PARAMETRIZADOS

@ParameterizedTest, @ValueSource

- Si el código de varios tests es idéntico a excepción de los valores concretos del caso de prueba que cada uno implementa, podemos sustituirlos por un test parametrizado.
- Se trata de implementar un único test, que anotaremos con `@ParameterizedTest`, y que tendrá como parámetros los valores concretos en los que se diferencian los tests a los que sustituye.
- Si el test parametrizado solamente necesita un parámetro, de tipo primitivo o String, usaremos la anotación `@ValueSource` para indicar los valores para ese parámetro

EJEMPLO de Test parametrizado usando `@ValueSource`

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

@ParameterizedTest
@ValueSource(strings = {"racecar", "radar", "able was I ere I saw elba"})
void palindromes(String candidate) {
    assertTrue(c.isPalindrome(candidate));
}
```

En este caso, los valores de la colección de parámetros son de tipo String

- El método `palindromes` es un tests parametrizado, con un parámetro de tipo String.
- El test se ejecuta 3 veces (con cada uno de los 3 parámetros indicados en `@ValueSource`.
- Otras alternativas posibles son `@ValueSource(doubles = {...})`, `@ValueSource(ints = {...})`, o `@ValueSource(longs = {...})`, dependiendo del tipo de dato del parámetro del test anotado con `@ParameterizedTest`

TESTS PARAMETRIZADOS

@ParameterizedTest, @MethodSource

EJEMPLO de Test parametrizado usando @MethodSource

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

@ParameterizedTest(name = "User {1}, when Alert level is {2} should
    NOMBRE MÉTODO have access to transporters of {0}")
@MethodSource("casosDePrueba")
void testParametrizado(boolean expected, Person user, Alert alertStatus) {
    transp.setAlertStatus(alertStatus);
    assertEquals(expected, transp.canAccessTransporter(user), invocación SUT
        () -> generateFailureMessage("transporter",
            expected, user, alertStatus));
}

//los valores devueltos por este método son los argumentos
//del método anotado con @ParameterizedTest
private static Stream<Arguments> casosDePrueba() {
    return Stream.of(
        Arguments.of(true, picard, Alert.NONE),
        Arguments.of(true, barclay, Alert.NONE),
        Arguments.of(false, lwaxana, Alert.NONE),
        Arguments.of(false, lwaxana, Alert.YELLOW),
        Arguments.of(false, q, Alert.YELLOW),
        Arguments.of(true, picard, Alert.RED),
        Arguments.of(false, q, Alert.RED)
    );
    expected user alertStatus
}

//método que construye y devuelve un mensaje de error en caso
//de que el resultado esperado no coincida con el real
private String generateFailureMessage(String system, boolean expected,
    Person user, Alert alertStatus) {
    String message = user.getFirstName() + " should";
    if (!expected) {
        message += " not";
    }
    message += " be able to access the " + system +
        " when alert status is " + alertStatus;
    return message;
}
```

El método `casosDePrueba()` devuelve un `Stream` con los Argumentos que se pasarán como parámetros al test parametrizado.
Cada objeto de tipo `Arguments` es un caso de prueba

Si el método anotado con `@ParameterizedTest`

requiere más de un parámetro, usaremos la anotación `@MethodSource` indicando un nombre de método.

El método `casosDePrueba` devuelve una colección de "tuplas" de valores (cada tupla representa una fila de la tabla de casos de prueba). El número de elementos de la tupla se corresponderá con el número de parámetros del test parametrizado.

El test parametrizado se invocará tantas veces como elementos de tipo `Arguments` tengamos. En el ejemplo serán 7 veces.

NO SE NECESITA EN LAS PRÁCTICAS

JUNIT 5 Y MAVEN

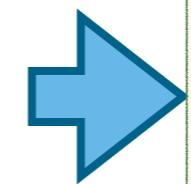
Para poder implementar y compilar de los tests

- Para poder implementar los tests con JUnit5 (y compilarlos) necesitamos incluir la librería "junit-jupiter-engine". De esta forma tendremos acceso las clases del paquete org.junit.jupiter.api, importándolas desde nuestro código de pruebas.

LIBRERIA JAVA QUE VIENE IDENTIFICADA POR COORDENADAS

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

junit-jupiter-engine5.7.0.jar



```
//código fuente de los tests en /src/test/java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Tag;
...
...
```

- El valor "test" de la etiqueta <scope> indica que la librería sólo se utiliza para compilar y ejecutar los tests. Por lo tanto NO podremos importar ninguna de sus clases desde /src/main/java
- Si usamos tests parametrizados, necesitaremos incluir también la libreria "junit-params" para poder usar las anotaciones correspondientes en nuestro código de pruebas.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

```
//código fuente de los tests en /src/test/java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;
...
...
```

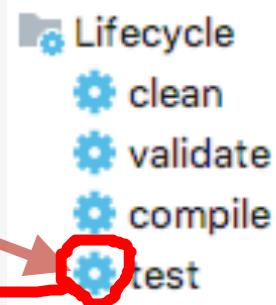


Modifica el pom SIEMPRE
manualmente!! No permitas que el
IDE añada/modifique el pom por tí!!

JUNIT 5 Y MAVEN

Para poder EJECUTAR los tests

<https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html>



- Ejecutaremos los tests mediante la goal `surefire:test`. El plugin `surefire` será, por tanto, el encargado usar las librerías JUnit5 que correspondan para ejecutar las pruebas
- Necesitamos incluir el plugin `surefire` en nuestro pom:

```
<plugin> → PROGRAMAS QUE EJECUTAN CIERTAS ACCIONES
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
</plugin>
```

PLUGIN
GOALS

mvn test

la goal `surefire:test` está asociada por defecto a la fase `test` de maven

- La goal `surefire:test` ejecuta todos los métodos anotados con `@Test` (o `@ParameterizedTest`) dentro de las clases cuyo nombre se corresponda con alguno de estos patrones: `**/Test*.java`, `**/*Test.java`, `**/*Tests.java`, `**/*TestCase.java`
- Para "filtrar" la ejecución de los tests en función de sus anotaciones `@Tag`, tendremos que configurar el plugin usando las propiedades "groups" y "excludedGroups" del plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <configuration>
    <groups>etiqueta1,etiqueta2</groups> GROUPS → EJECUTAR
    <excludedGroups>excluidos</excludedGroups> EXCLUDEDGROUPS → NO EJECUTAR
  </configuration>
</plugin>
```

mvn test

En este ejemplo se ejecutarán los tests etiquetados como "etiqueta1" y también los etiquetados con "etiqueta2". También podemos excluir una (o varias etiquetas) del filtro

Sí algún test falla, el proceso de construcción se detiene y se obtiene un BUILD FAILURE!!

CONFIGURACIÓN DEL PLUGIN SUREFIRE

Se puede configurar cualquier plugin

<https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html>

- Podemos alterar el valor de ciertas propiedades de los plugins usando la etiqueta **<configuration>**. Cada plugin tiene su conjunto de propiedades. La única forma de saber cómo usar correctamente un plugin es acceder a su documentación.
- Como ya hemos visto, podemos filtrar la ejecución de los tests a través de sus etiquetas, usando las propiedades **groups** y/o **excludedGroups** del plugin surefire, en el pom de nuestro proyecto Maven REPETIDO
- De forma alternativa, podemos configurar cualquier plugin desde línea de comandos:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
</plugin>
```



mvn test

Usamos la configuración por defecto en el pom, y la cambiamos desde línea de comandos

-Dgroups=etiqueta1,etiqueta2 -DexcludedGroups=excluidos

- Debes tener en cuenta de que si configuramos el plugin en el pom y también desde línea de comandos, la configuración del pom PREVALECE sobre la de línea de comandos. Si por ejemplo, quisieramos ejecutar siempre un cierto subconjunto de tests y ocasionalmente otro subconjunto, podríamos hacer esto:

```
<properties> <filtrar.excluidos> importates, fase1 </filtrar.excluidos>
  <filtrar.por>importantes, fase1</filtrar.por>
</properties>
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <configuration>
    <groups>${filtrar.por}</groups>
  </configuration>
</plugin>      <excludedGroups> ${filtrar.excluidos}</excludedGroups>
```

ELI



mvn test

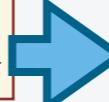
Se ejecutan los tests etiquetados como "importantes" más todos los tests etiquetados como "fase1"



mvn test -Dfiltrar.por=rapidos

Se ejecutan los tests etiquetados como "rapidos"

mvn test -Dfiltrar.excluidos=lentos -> No se ejecutan los etiquetados como "lentos"



mvn test -Dfiltrar.por=""

Se ejecutan todos los tests

INFORMES JUNIT

Ejecutaremos los tests desde la ventana Maven!!!

- P O Cuando ejecutamos cada test, podemos obtener 3 resultados posibles:
 - **Pass**: cuando el resultado esperado coincide con el real
 - **Failure**: el método `Assert` lanza una excepción de tipo `AssertionFailedError`
 - **Error**: se genera cualquier otra excepción durante la ejecución del test PROBABLEMENTE NO LLEGA NI AL ASSERT
- P O El **informe** de la ejecución de todos los tests se guarda en `target/surefire-reports`, en formatos `txt` y `xml` (un informe por clase)
 - Dependiendo de la herramienta que "lea" dichos informes, éstos se mostrarán de forma diferente

```
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   CalculadoraTest.suma1:22 expected: <1> but was: <0>
[ERROR]   CalculadoraTest.suma2:28 expected: <7> but was: <4>
[ERROR]   CalculadoraTest.suma4:40 expected: <13> but was: <5>
[ERROR] Errors:
[ERROR]   CalculadoraTest.suma3:34 » Arithmetic / by zero
[INFO]
[ERROR] Tests run: 5, Failures: 3, Errors: 1, Skipped: 0
```

Informe de pruebas
Maven

Run 'CalculadoraTest'

Test Results		49 ms
!	CalculadoraTest	49 ms
×	sum1()	38 ms
×	sum2()	2 ms
!	sum3()	2 ms
×	sum4()	1 ms
✓	sum5()	6 ms

Desde el menú
contextual de
CalculadoraTest.java

Sesión 2: Drivers

Informe de pruebas IntelliJ

Sólo muestra los
informes de la última
ejecución de la fase
test de Maven

!	ppss.CalculadoraTest
×	sum1
×	sum2
!	sum3
×	sum4
✓	sum5

Informe obtenido por
el plugin "Maven Test
Results" de IntelliJ, a
partir del informe de
Maven

MAVEN Y PRUEBAS UNITARIAS

P

P

Sesión 2: Drivers

Fase compile:

Se compilan los fuentes del proyecto (/src/main/java)

GOAL por defecto: **compiler:compile**

artefactos generados en /target/classes

Si hay errores de compilación se detiene la construcción.

Por defecto



Fase test-compile:

Se compilan los tests unitarios (/src/test/java)

GOAL por defecto: **compiler:testCompile**

artefactos generados en /target/test-classes

Si hay errores de compilación se detiene la construcción.

Por defecto



Fase test:

Se ejecutan los tests unitarios (/target/test-classes)

Se ejecutan los métodos anotados con @Test de las clases

**/Test*.java, **/*Test.java, o **/*TestCase.java

GOAL por defecto: **surefire:test**

artefactos generados en /target/surefire-reports.

por defecto

Se detiene el proceso de construcción si algún tests falla.

Default lifecycle

validate

initialize

generate-sources

process-sources

generate-resources

process-resources

compile

process-classes

generate-test-sources

process-test-sources

generate-test-resource

process-test-resources

test-compile

process-test-classes

test

prepare-package

package

pre-integration-test

integration-test

post-integration-test

verify

install

deploy

Y AHORA VAMOS AL LABORATORIO...

P

Vamos a automatizar el diseño de casos de prueba que hemos obtenido en prácticas anteriores

P

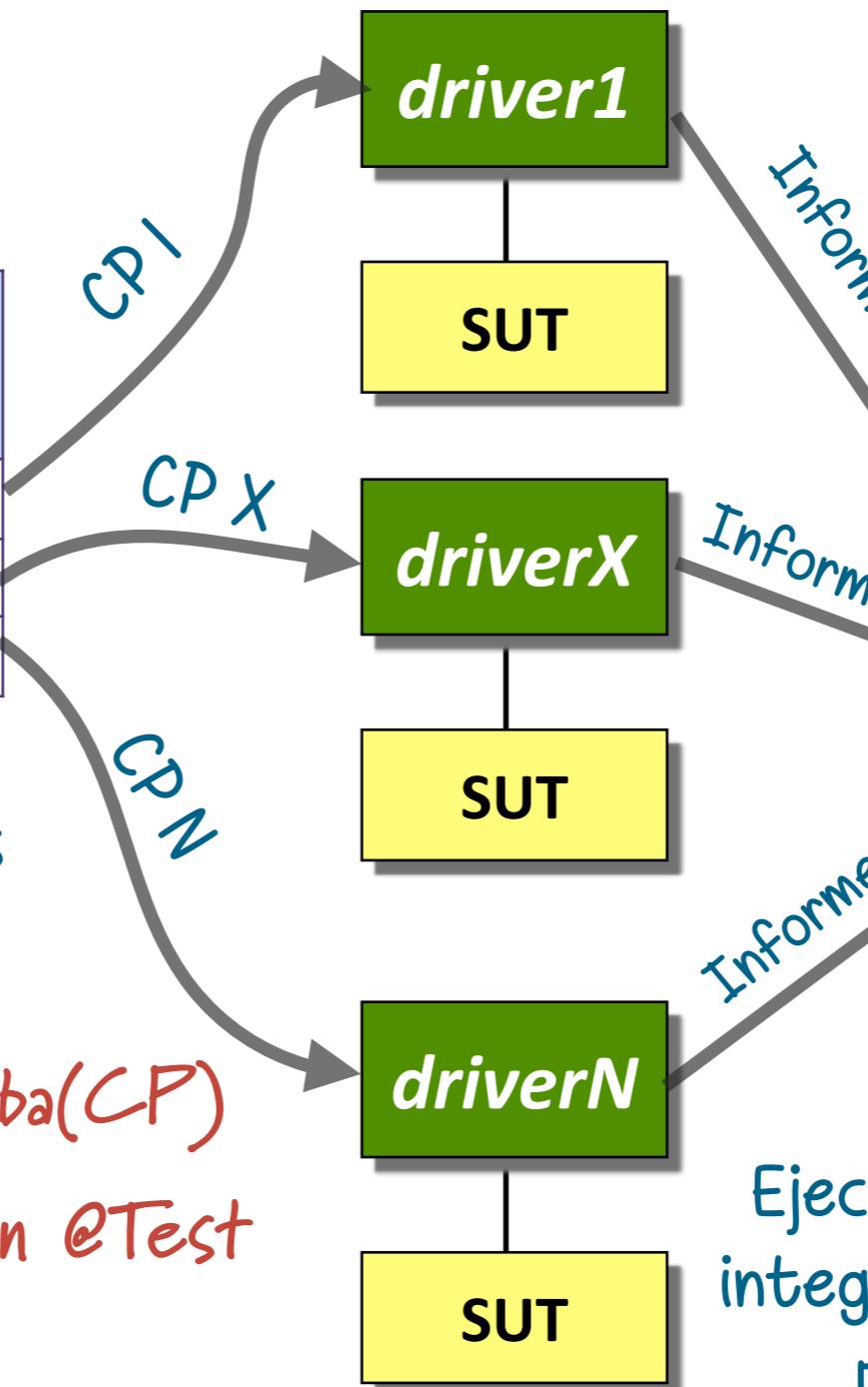
tabla de casos de prueba

Dato Entrada 1	Dato Entrada 2	Dato Entrada k	Resultado Esperado
d11=...	d21=...	dk1=...	r1
d1n=...	d2n=...	dkn=	r n

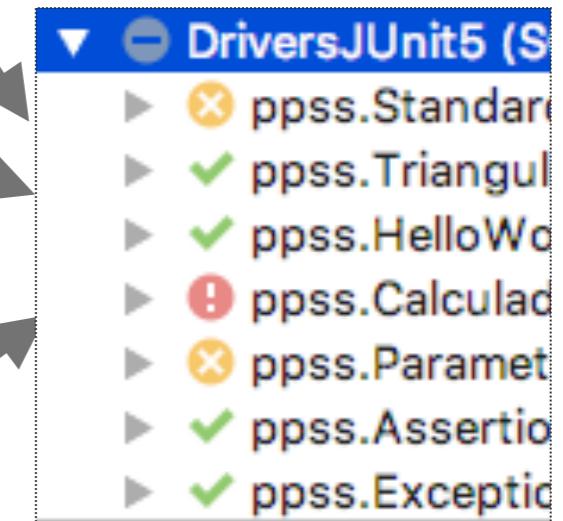
Implementaremos los drivers
utilizando JUnit 5

1 driver x cada caso de prueba(CP)

1 driver = método anotado con @Test



mvn test



Ejecutaremos los tests JUnit
integrándolos en el ciclo de vida
por defecto de Maven

PREFERENCIAS BIBLIOGRÁFICAS

- P
 - Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulo 3: Unit Testing
 - JUnit 5 (<https://junit.org/junit5/docs/current/user-guide/>)
 - Annotations
 - Assertions
 - Tagging and Filtering
 - Parameterized Tests
 - Running tests
 - What's new in JUnit 5? (Blog Scott logic, 10 octubre 2017)
 - Java Lambda Expressions (tutorials.jenkov.com, 18 enero 2019)
 - Java Lambda Expressions Basics (Dzone, java zone, 2013)